RCE: An Integration Environment for Engineering and Science

Brigitte Boden, Jan Flink, Robert Mischke, Kathrin Schaffert,
Alexander Weinert, Annika Wohlan, and Andreas Schreiber

German Aerospace Center (DLR), Simulation and Software Technology,
Linder Höhe, 51147 Köln, Germany
firstname.lastname@dlr.de

Abstract. We present RCE (Remote Component Environment), an open-source framework developed primarily at DLR that enables its users to construct and execute multidisciplinary engineering workflows comprising multiple disciplinary tools. To this end, RCE supplies users with an easy-to-use graphical interface that allows for the intuitive integration of disciplinary tools. Users can execute the individual tools on arbitrary nodes present in the network and all data accrued during the execution of the workflow are collected and stored centrally. Hence, RCE makes it easy for collaborating engineers to contribute their individual disciplinary tools to a multidisciplinary design or analysis, and simplifies the subsequent analysis of the workflow's results.

Keywords: Multidisciplinary Analysis, Tool Integration, Workflow Execution, Collaboration, Distributed Execution

1 Motivation and Significance

Designing and evaluating modern systems requires collaboration among experts from a wide range of disciplines. Each of these experts uses highly specialized software tools to produce or investigate a design artifact such as the shape of a workpiece, its aerodynamic properties, or the cost for producing it. Orchestrating the runs of these tools as well as collecting, distributing, and archiving all intermediate data imposes a significant organizational overhead on the design process. RCE (Remote Component Environment) enables engineers and scientists to construct automated workflows consisting of numerous such software tools, to execute these workflows on a distributed network of compute nodes, and to collect all relevant artifacts for further analysis.

As an example, consider a team of engineers working to determine whether to construct a novel airplane wing out of steel, aluminum, or carbon. There is also a project leader who coordinates the activities involved in determining the optimal material. To this end, for each of the three materials, the engineers compute both the effectiveness of the resulting wing as well as its production cost. For the sake of simplicity, we assume the effectiveness of a wing shape is determined solely based on the lift it produces.

Each of the above materials may induce different constraints on the possible shapes. Hence, the project leader first obtains a description of these constraints from the collaborating material scientist. She then passes these constraints on to a team of engineers that determine, for each material, an optimal feasible shape that produces maximal lift. They do so by iteratively designing wing shapes and evaluating the lift produced by the respective shape, thus implementing a simple optimization loop. Having determined an optimal wing shape for a given material, the project leader then forwards the optimal design to another department that estimates the production cost. We illustrate the flow of information between the participants involved in the design of the wing in Figure 1.

Each step of such a design process uses one or multiple specialized software tools. Without a common software to coordinate data management, input and output files are typically transmitted between these tools using ad hoc methods like email, flash drives, or shared network drives. Even within a single design process, both the programming languages used for implementing these tools as well as the environments required for executing them are often heterogeneous. In our example, engineers may compute the shape constraints using Python [13] running on a desktop computer, whereas the wing shape may be constructed using Matlab [9] on a compute cluster. Finally, the aerodynamic evaluation may be performed using a simulation implemented in Fortran [2] using a GPU cluster while the estimate of production costs may be implemented in Java [16].

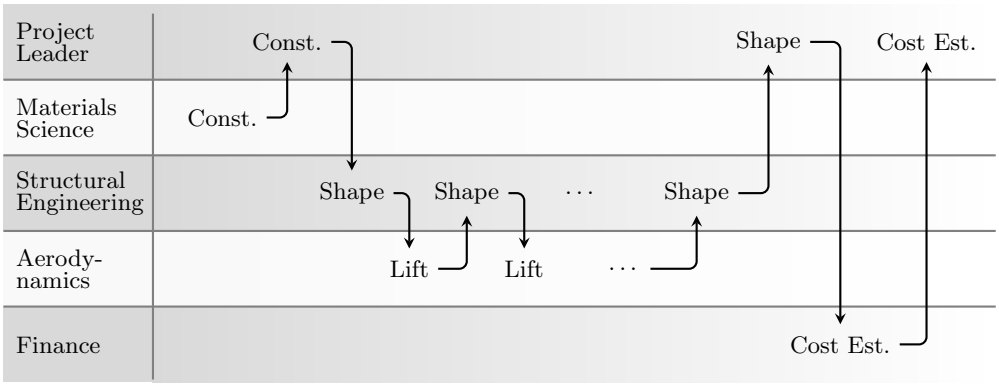

Fig. 1. Workflow for designing and evaluating a wing shape.

There are several obstacles in executing such a multidisciplinary workflow. Not only do the participants from different disciplines have to agree on methods for collaborating, possibly across the boundaries of business units, but also, on a more technical level, on data formats for data exchange. Moreover, they have to manually invoke their respective tools as well as subsequently collect, disseminate, and archive the resulting data. Even if this process is automated at all, this automation usually consists of custom solutions that are not easily reusable for other workflows.

RCE is an open-source software that enables scientists to overcome these recurrent obstacles when executing workflows involving multidisciplinary software tools running in diverse runtime environments. To do so, RCE lets the user grant other users in the same network access to selected tools on their local machine, allowing the remote users to execute the tool on demand, while the tool owner retains full control over the implementation files and runtime environment of the tool. RCE furthermore enables the users to construct and execute automated workflows comprised of both local tools as well as those published by other users in the same network. During automated workflow execution, RCE requires no further user input and orchestrates both the distribution of inputs and outputs between the involved tools and their invocation on the respective machines. Finally, it allows users to monitor the artifacts and console output accrued during the execution of the individual tools. All execution data is also automatically persisted, allowing access to and analysis of this data both during and after a workflow's execution.

In doing so, RCE facilitates cooperation between engineers and researchers of widely differing fields, enables them to leverage their domain expertise to simulate and analyze complex systems. RCE furthermore supports the engineers and researchers in discovering and tracing relationships between the input parameters of the workflow and its results.

2 Software Description

We first describe the use cases supported by RCE in Section 2.1, before we discuss the implementation of the features enabling these use cases in Section 2.2.

2.1 Software Functionalities

RCE is a general tool for engineers and scientists from a wide range of disciplines to create and execute distributed workflows as well as evaluate the obtained results. Hence, RCE supports a wide range of use cases.

As a first use case, consider a single engineer who has to orchestrate the execution of multiple tools in her daily work and wants to automate this process. To this end, she can integrate external tools for calculations, simulations, and evaluations using RCE. RCE provides a graphical wizard that guides the user through the tool integration process.

During integration, the user defines inputs and outputs of the tool using the native datatypes of RCE, which comprise typical primitive data types such as Booleans, integers, and floating point numbers as well as more general ones such as files and directories. The user can also define script sections that are

executed before and after each tool run (called pre- and post-scripts), which is typically used for the conversion of input transmitted by RCE into tool-specific layouts and, vice versa, for the conversion of tool-specific output data into more common formats before they are handed over to RCE. Finally, the user defines the commands to execute the underlying tool given the previously defined inputs.

We illustrate the integration of a tool into RCE in Figure 2. In this example, the pre-script takes one parameter that is supplied by RCE and provides a second, hard-coded parameter. The integrated tool is then called with these two parameters and provides three output values. Out of these output values, the post-script discards one and passes the other two back to RCE.

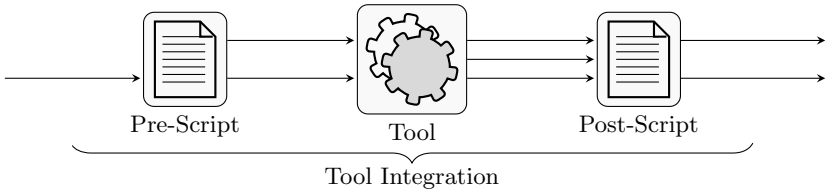

Fig. 2. A conceptual view of the integration of a tool into RCE.

In addition to the user-integrated tools, RCE provides a number of predefined tools which supply basic and often-needed functionalities such as controlling the flow of data through the workflow, reading and extracting data from XML files, executing user-supplied Python scripts, as well as a number of tools that provide mathematical and statistical methods to evaluate the incoming data. Furthermore, RCE supplies components for the construction of design-, evaluation- and optimization loops, most prominent among them an integration of the well-known and versatile Dakota framework [1] for optimization. The user can mix and match both pre-integrated tools and user-integrated ones to construct complex workflows in the graphical workflow editor of RCE. As an example, a user may construct a workflow for determining the optimal material for the construction of an airplane wing as described in Section 1. We show such a workflow in Figure 3.

After constructing the workflow, the user can start its execution. As a first step, RCE triggers the execution of all tools without inputs. Once a tool has terminated, RCE collects its outputs as defined by its integration and computed by its post-script, and passes them on to the tools whose inputs are connected to the outputs of the terminated tool. RCE then executes the succeeding tools. Once it has collected all inputs defined by the integration of a tool, it executes the respective tool and again collects its outputs to pass them to subsequent tools. This is continued until no tool has produced any further output data, which is typically the case either because a linear workflow has executed all of its tools, or because a main evaluation loop has reached a certain end criterion (e.g., convergence).

Execution via RCE is also possible for tools that are under active development. Instead of "freezing" a certain version of software tools and executing this version every time the workflow is executed, RCE instead uses the version currently installed on the machine. Hence, there is no additional overhead in deploying new versions of actively developed tools to RCE. Instead, RCE executes the tool version that has been most recently deployed. Once such a tool has stabilized, the user may provide both a "development" version and a "stable" version in parallel. Thus, one set of users may use the stable version of a tool, while others gain immediate access to one or more development versions at the same time.

Now consider the use case that the user wants to execute some parts of the workflow on her local machine, while she wants to execute others on a compute cluster or some remote cluster equipped with GPUs. To this end, she can connect her local instance of RCE with one or multiple instances running on remote machines reachable via a local network or the Internet. She may configure the remote instances either via a graphical interface, if the remote machine offers one, or via the built-in command-line interface of RCE, which is also reachable directly via an SSH connection.

The engineer may also connect to instances of RCE controlled by other engineers via the same mechanism. Users can publish their tool integrations via the network that was created this way, allowing remote users to use the locally integrated tools in their workflows. When constructing a workflow, remote and local components appear side-by-side in the graphical editor and are indistinguishable from one

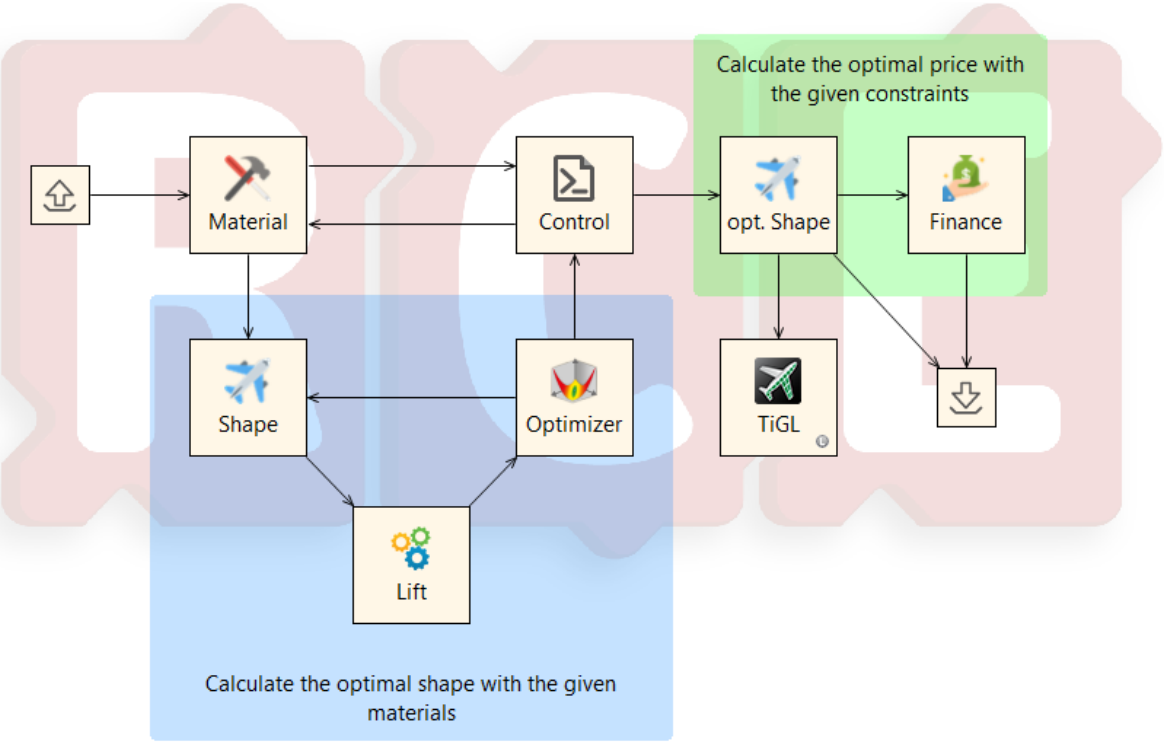

Fig. 3. A workflow in RCE.

another. This allows the user to construct the logic of the workflow without having to determine its execution configuration at the same time.

For execution of the resulting workflow comprising both local and remote components, the latter ones are not copied to the machine executing the workflow, but they are instead executed on the machine on which they are integrated. RCE supplies them with the required input data via the network connection, retrieves their output after termination and transmits that data to the machine coordinating the workflow's execution.

In order to simplify the setup of RCE networks for collaboration, RCE instances can also be configured to run as a so-called relay. Other RCE instances can then connect to this instance, which acts as an intermediate, forwarding relevant information between all instances as needed. As RCE's network structure is decentralized, any number of relays can be combined, which is often used to first connect the RCE instances within a working group, and to then assemble these groups into larger networks.

If the engineer using RCE has integrated all tools involved in her workflow onto remote instances, she may want to shut down her local machine during the execution of the workflow. To this end, upon execution of a workflow, the user may designate some instance to be the workflow controller, which orchestrates the distribution of data among the involved instances of RCE. This is particularly useful for the execution of long-running workflows, as the user may start the workflow on their local machine and designate a server as the workflow controller, allowing them to shut down their local machine after the start of the workflow.

Finally, consider the case that the engineer wants to inspect the output of individual executions of some particular tool that has been invoked during the execution of the workflow. To this end, all data, i.e., all inputs to and outputs from all components accrued during a workflow run are collected by the workflow controller and can be monitored both during execution of the workflow and after its completion by all RCE instances connected to the workflow controller. This provides a huge benefit for the collaborative analysis of the results of the workflow over the existing ad-hoc dissemination of such results.

In practice, the roles of the RCE instances involved in the execution of a workflow may not be as clearly separated as in the use cases discussed above. A single RCE instance can be configured to display

a user interface for the construction of workflows, to publish integrated tools and act as a workflow controller, and to merge connected networks, or to do any combination of these tasks.

2.2 Software Architecture

The user base of RCE is mainly comprised of engineers and scientists who are not primarily interested in designing workflows, but rather in performing multidisciplinary analyses. Hence, we provide users with a software package that is *(i) easy to deploy and configure* and *(ii) features very few external dependencies*. Moreover, such analyses are usually performed on specialized hardware that is shared with other users, e.g., a compute cluster. Hence, the users of RCE may not have complete administrative control over the machine they use. Thus, another main driver behind the architectural decisions regarding RCE is the requirement that the resulting software must be able to *(iii) run on a wide range of machines with (iv) minimal privileges*. As RCE is not developed for use in a single domain, but rather as a general tool for the development and execution of workflows, we require a *(v) modular architecture* that is amenable to the development of additional modules as required by certain projects. Finally, since users must be able to configure networks of RCE instances, we require an architecture that allows for *(vi) easy communication* between connected instances.

To achieve platform independence, we opted to implement RCE in Java [16]. As we furthermore rely on the platform-independent Eclipse Rich Client Platform (RCP) [5], we can provide executable artifacts for Windows and Linux while only maintaining a single code base. By also distributing RCE as a simple zip file (besides more specialized installation packages), which contains all dependencies of RCE save for the actual Java runtime, we achieve the first, second, and third requirement listed above. As RCE only requires a normal user account without elevated privileges, we furthermore achieve the fourth requirement above.

To achieve the fifth requirement above, RCE has a highly modular structure on the source code level. Every significant functionality is accessible from other functional units via an explicit service interface. This approach ensures a clear separation of code segments, improves maintainability, and facilitates testing. We define these services via the OSGi Declarative Services standard [11], and manage them at runtime using the Eclipse Equinox OSGi implementation [4] included in RCP.

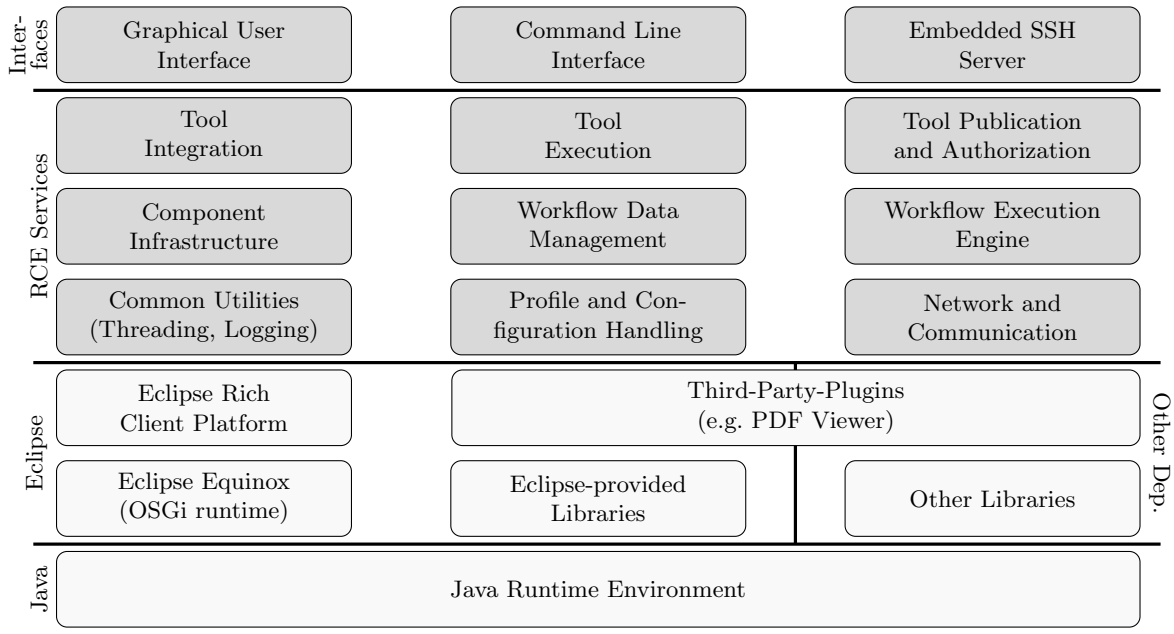

Fig. 4. A top-level view of the services implemented in RCE.

As of version 9.1, RCE has more than 230 service interfaces. Due to space limitations, we only present an overview of its major functional groups in Figure 4. In that figure, dark gray boxes denote the service

groups implemented as part of RCE, whereas lightly shaded boxes denote external dependencies. These external dependencies include the Java runtime itself, but also libraries provided by the Eclipse Rich Client Platform, as well as other third-party-libraries. In addition to the services shown in Figure 4, RCE also contains general utilities for authorization, multithreading, logging, testing, installation of updates, and management of multiple user-defined profiles.

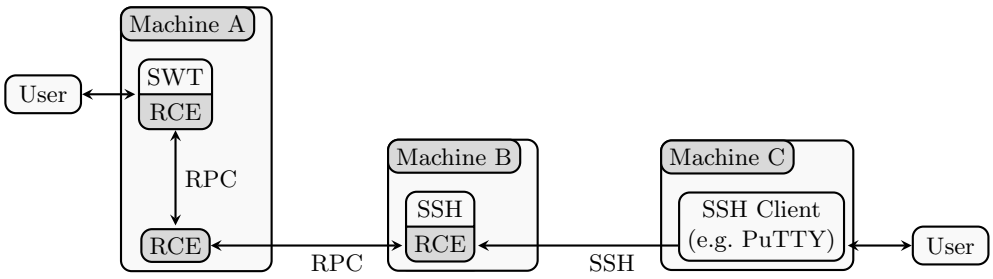

Fig. 5. A typical setup of RCE instances and the communication between them.

For communication between different instances, we also use service interfaces to define remote procedure calls (RPCs), which help us in achieving the final requirement above. On a source code level, a call to a remote procedure is indistinguishable from a call to a local one, thus reducing complexity for software developers. In Figure 5 we present an example of the deployment and usage of RCE as well as the communication between the individual instances.

3 Concrete Example

Consider again the example of the design of an airplane wing from Section 1 and recall that the individual tools implementing the steps of the process as illustrated in Figure 1 may be required to run on different machines. Although these machines may be part of different disjoint networks, we assume that the networks themselves are connected via the Internet. Otherwise, no communication is possible. We illustrate such a network setup in Figure 6.

In that drawing, the vertices labeled MS, AD, F, and SE denote machines under the control of the department for materials science, aerodynamics, finance, and structural engineering, respectively. Furthermore, the vertex labeled R represents a dedicated relay server in a network location that is reachable from all project participants. We denote these isolated networks by dark gray boxes in Figure 6, each of which has an interface connecting it to the relay server.

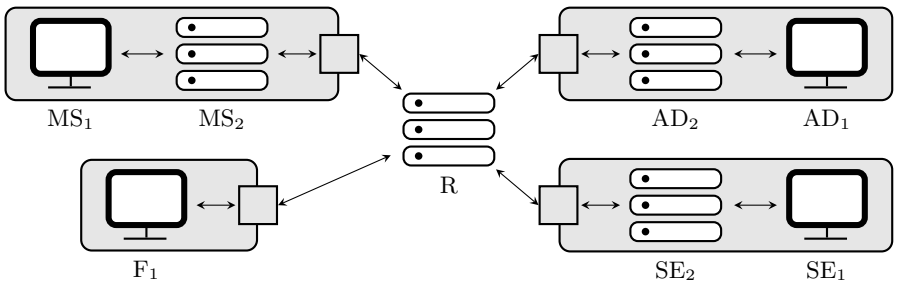

Fig. 6. A network setup of the machines taking part in the process shown in Figure 1.

Each involved project partner starts by defining the integration of their respective tool, as described in Section 2.1, on their local computer (i.e., on the machines MS_1, AD_1, SE_1, and F_1). This may happen either using the graphical wizard for tool integration, or via manually editing the files defining the tool integration. After completing and testing the integration locally, the users move the integration to the

compute nodes (i.e., to the machines labelled MS_2, AD_2, and SE_2). For simpler tools, in contrast, the users may opt to leave both the tool as well as the integration on their local computers. In our example, this is the case for the finance department, which does not use a compute node.

Each machine that hosts one or more tools is connected to the central relay server R, thus making them visible to one another. The users then collaborate in constructing the workflow shown in Figure 3, which formalizes the workflow illustrated in Figure 1 on Page 2. Within this workflow, they can transparently mix tool instances that were moved to a compute node with tools that are still published from a user's machine.

Having finished the construction of the workflow, the users may execute it using, e.g., the relay server or one of the compute nodes as the controller for the distributed execution. Using such a shared server as the workflow controller allows machines not included in the actual computation (i.e., the machines MS_1, AD_1, and SE_1) to be safely turned off or disconnected from the RCE network during the execution of the workflow. This is particularly useful for machines that are not permanently attached to the same physical network, e.g., work laptops switching between cable and wireless adapters.

4 Impact

RCE is widely used among many projects at DLR involving users from the fields of aeronautics and astronautics, as well as traffic and energy. Since a complete overview over all projects that involve RCE is impossible due to its nature as freely distributed open-source software, we highlight a number of representative projects here.

A number of aerospace-projects use RCE. Boden, Flink, Mischke, et al. [3] give an overview over such projects. Here, we highlight two such projects, namely FrEACs and Digital-X, as well as a number of other projects that leveraged the capabilities of RCE.

4.1 FrEACs

The goal of the project FrEACs [10] was to design a product development process for aircraft configurations. To this end, the engineers involved defined engineering services, i.e., disciplinary tools that were required by other participants in the project. Both the publication as well as the connection of the published engineering services were done via RCE. Since, in this development setting, each engineering service was owned and maintained by a specialist engineer, the project benefited greatly from the use of RCE, as each service and its executions remained under complete control of their owners. The resulting data was organized using the integrated project explorer of RCE, which allowed for quick analysis of the data generated by running the individual services.

A major part of the project FrEACs consisted of regularly occurring design camps, in which all project participants gathered in a single location to further design and enhance the development process. These design camps not only greatly benefited from RCE's ability to export the accrued data already during the execution of the workflow, but also from the easy access to the data from all instances of RCE that are connected to the network. This allowed the users to detect and fix errors early in the design phase of the workflow.

4.2 Digital-X

RCE was also used in the DLR project Digital-X [7, 8]. While previously, RCE workflows were mainly used in the low-fidelity pre-design of aircraft, the aim of this project was the development of a multi-disciplinary optimization process that comprised both low- and high-fidelity simulations of aircraft. To this end, a number of highly specialized tools running on either Windows or Linux were integrated into a single workflow. These tools were contributed by eight DLR institutes and hosted at six different locations. Hence, the use of RCE in this project showcases the ease of integrating tools requiring different runtime environments into a single workflow.

Moreover, although starting the high-fidelity analysis required the results of the low-fidelity analysis, the individual steps of the former and the latter analyses were partly independent of previous results. Thus, the performance of these individual steps improved markedly over previous implementations of this workflow, as RCE automatically executes independent tools in parallel [7].

4.3 PEGASUS

Apart from its use in the design of complete aircraft, RCE was also employed for the design of individual airplane engines as part of the project PEGASUS [14]. In this project, a special integration interface was developed that allowed an existing C++ software to connect to RCE instances using the standardized SSH protocol. Leveraging RCE's existing network capabilities, this allowed that software to execute distributed simulation tools, and also invoke other instances of the C++ software itself. The latter was also done to improve performance by distributing compute loads across multiple machines, which required system information to do load balancing. To support this, RCE made the performance data that it already collects available over the SSH interface. To simplify usage of these features, all communication with RCE was encapsulated in a client library providing a C API, which allows it to be used from a wide range of programming languages. This library is currently being modernized and is planned for inclusion in future RCE releases.

4.4 TRIAD

In addition to its use in numerous aerospace projects, RCE was furthermore used in the project TRIAD [17] which focused on developing a design environment for helicopters. Here, again multiple individual disciplinary tools were integrated into a single workflow comprising both low- and high-fidelity analysis and optimization of rotorcraft. Since the aim of the project was not to develop a single rotorcraft, but rather to provide a general environment with which multiple such craft could be designed, there was no fixed workflow to be implemented. Instead, the participating engineers integrated their tools into RCE and published their tools onto the network.

One of the major work items of this project was the development of the actual workflow. As its structure was not clear at the beginning of the project, RCE's intuitive graphical interface significantly reduced the workload of the individual engineers. Moreover, several of the involved tools were proprietary and were restricted to running on the machines of certain developers. Hence, RCE's capability of executing tools on the publishing machine itself proved to be a great benefit to the construction of the overall workflow.

4.5 CLAVA

Finally, while a large number of known users works in aerospace engineering, RCE is also deployed in the field of astronautics. Prime among these applications is the use of RCE in the project CLAVA [6], which has the task of developing the next generation of launchers. This design process is highly dynamic in nature, since the used tools are regularly replaced by newer versions or other tools entirely. Hence, the involved engineers were able to speed up the design process via the use of RCE, which transparently uses new versions of the tools as soon as they are deployed and allows for easy replacement of the used tools. Furthermore, another increase in velocity resulted from the automatic execution of the workflow which was previously executed manually, i.e., via manual execution of the involved tools and dissemination of their outputs.

For the sake of readability, we omit detailed descriptions of other projects in which RCE is deployed or currently involved. These projects involve ones from the domains of logistics and climate research (e.g., TraK or VEU II [15]), energy research (e.g., INTEEVER II), or shipbuilding (e.g., HOLISHIP [12]). Moreover, since RCE is freely available as open source software, we do not have access to a comprehensive list of users.

5 Conclusions

In this work we have presented RCE (Remote Component Environment), an open-source integration environment for engineers and scientists that aids them in the multidisciplinary development, analysis, and the optimization of complex systems. We have given an overview over the main features of RCE and argued that due to the richness of this feature set, RCE is able to adapt to numerous usage scenarios. Instead of having to adapt to a fixed design process prescribed by RCE, engineering teams can use RCE to enhance their existing design processes. Subsequently, we have outlined the technical implementation

of these features and we have shown that our choice of frameworks, development methodology, and supporting technologies enables not only the further maintenance of RCE, but also the development of additional features as required by future projects.

Finally, we have given an overview over research and development projects at DLR that use or have used RCE as part of their engineering processes. These projects showcase the versatility of RCE, as it is not only used in a number of domains, but since it also supports a wide range of scales and scopes of projects. This includes both low- and high-fidelity analyses, simulations, and optimizations, as well as different modes of interaction, ranging from iterative design work using the graphical interface up to completely automated invocations of RCE via its network interfaces.

For future work, we will further improve the usability of RCE and increase its feature set based on requirements arising from the use in scientific and engineering projects. Prime among these future developments is an extension and modularization of the central code governing the execution of workflows, which forms one of the oldest and most monolithic parts of the current codebase. Further, we are adapting RCE for use in cloud environments to allow users to optimally leverage shared resources. Moreover, as there is an increasing need for sharing integrated tools between cooperating organizations, dedicated support for transfer of sensitive data over insecure communication channels, such as the Internet, will be added in the near future. To address the IT security demands of such setups, the network protocol for this will be encrypted, authenticated, and restricted to a minimal and easily audited feature set. Finally, due to the ever-increasing size of workflows, one of our main goals is to provide users with improved editing and management features for larger workflows. This includes both the capability to modularize parts of workflows into sub-workflows, as well as the ability to publish complete workflows as virtual tools, which can then be used in other workflows just like standard tools. Together, these features will improve the design as well as simplify testing and maintenance of large-scale workflows.

Acknowledgements We gratefully acknowledge the support and contributions of previous team members Doreen Seider, Arne Bachmann, Tobias Brieden, Markus Kunde, Markus Litz, Oliver Seebach, Heinrich Wendel, and Sascha Zur. Furthermore, we would like to thank all other colleagues, students, and users that have contributed to the development of RCE.

References

1. Adams, B., Bauman, L., Bohnhoff, W., Dalbey, K., Ebeida, M., Eddy, J., Eldred, M., Hough, P., Hu, K., Jakeman, J., Stephens, J., Swiler, L., Vigil, D., , Wildey, T.: Dakota, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 6.0 Users Manual. Sandia Technical Report SAND2014-4633 (2014), available at `https://dakota.sandia.gov/content/manuals`
2. Backus, J., IBM, ISO: The Fortran Programming Language
3. Boden, B., Flink, J., Mischke, R., Schaffert, K., Weinert, A., Wohlan, A., Ilic, C., Wunderlich, T., Liersch, C.M., Goertz, S., Ciampa, P.D., Moerland, E.: Distributed Multidisciplinary Optimization and Collaborative Process Development Using RCE. In: AIAA Aviation 2019 Forum. American Institute of Aeronautics and Astronautics. https://doi.org/10.2514/6.2019-2989
4. Eclipse Foundation: Eclipse Equinox, `https://www.eclipse.org/equinox/`
5. Eclipse Foundation: Rich Client Platform, `https://wiki.eclipse.org/Rich_Client_Platform`
6. Fischer, P.M., Deshmukh, M., Koch, A.D., Mischke, R., Gomez, A.M., Schreiber, A., Gerndt, A.: Enabling a Conceptual Data Model and Workflow Integration Environment for Concurrent Launch Vehicle Analysis. In: IAC 2018. `https://elib.dlr.de/124541/`
7. Görtz, S., Ilić, C., Jepsen, J., Leitner, M., Schulze, M., Schuster, A., Scherer, J., Becker, R., Zur, S., Petsch, M.: Multi-Level MDO of a Long-Range Transport Aircraft Using a Distributed Analysis Framework. In: 18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference. American Institute of Aeronautics and Astronautics (2017). https://doi.org/10.2514/6.2017-4326
8. Görtz, S., Ilic, C., Abu-Zurayk, M., Liepelt, R., Jepsen, J., Führer, T., Becker, R.G., Scherer, J., Kier, T., Siggel, M.: Collaborative Multi-Level MDO Process Development and Application to Long-Range Transport Aircraft. In: ICAS 2016. International Council of the Aeronautical Sciences
9. MathWorks, Inc.: MATLAB, `https://www.mathworks.com/products/matlab.html`
10. Moerland, E., Pfeiffer, T., Bhnke, D., Jepsen, J., Freund, S., Liersch, C.M., Chiozzotto, G.P., Klein, C., Scherer, J., Hasan, Y.J., Flink, J.: On the Design of a Strut-Braced Wing Configuration in a Collaborative Design Environment. In: 17th AIAA Aviation Technology, Integration, and Operations Conference. American Institute of Aeronautics and Astronautics (2017). https://doi.org/10.2514/6.2017-4397

11. OSGi Alliance: OSGi Compendium Release 7 - Declarative Services Specification, https://osgi.org/specification/osgi.cmpn/7.0.0/service.component.html
12. Papanikolaou, A. (ed.): A Holistic Approach to Ship Design. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-02810-7
13. Python Contributors: The Python Programming Language, available at https://www.python.org
14. Reitenbach, S., Krumme, A., Behrendt, T., Schnös, M., Schmidt, T., Hönig, S., Mischke, R., Moerland, E.: Design and Application of a Multi-Disciplinary Pre-Design Process for Novel Engine Concepts. Journal of Engineering for Gas Turbines and Power **141**(1) (2018). https://doi.org/10.1115/1.4040750
15. Seum, S., Ehrenberger, S., Kuhnimhof, T., Winkler, C.: Verkehr und seine Umweltwirkungen. Internationales Verkehrswesen (2), 49–53 (2019)
16. Sun Microsystems: The Java Programming Language
17. Weiand, P., Buchwald, M., Schwinn, D.: Process Development for Integrated and Distributed Rotorcraft Design. Aerospace **6**(2) (2019). https://doi.org/10.3390/aerospace6020023